1981

Conversion from Data-Flow to Synchronous Execution in Loop Programs

Janice D. Cuny

Lawrence Synder

Report Number:
81-392

Conversion from Data-flow to Synchronous Execution in Loop Programs

Janice E. Cuny
Lawrence Snyder

Purdue University

CSD-TR-392

ABSTRACT

We present conversion algorithms that would enable programmers to program in a high-level, data-flow language and then run their programs on a synchronous machine. A model of interprocess communication systems is developed in which both data-flow and synchronous execution modes are represented. For a subclass of parallel programs, called loop programs, we characterize the programs for which conversions are possible in terms of sets of balancing equations. We show that all loop programs having the finite buffer property can be converted into synchronous mode. Finally two algorithms for the conversion of loop programs are presented and discussed.

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO-100.

Conversion from Data-flow to Synchronous Execution in Loop Programs

Janice E. Cuny

Lawrence Snyder

Purdue University

The preparation of highly parallel programs is not yet a routine programming activity. When we compare it to sequential programming where there are numerous general problem solving techniques, extensive programming language and system support, and a large corpus of thoroughly analyzed and tested algorithms and data structures, parallel programming is presently at a very primitive stage of development.

One difficulty of course, is synchronization - making sure that the right processor processes the right data at the right time. The synchronization problem can apparently be simplified by use of a data-driven or data-flow based execution mode. In this mode, each processor idles in a busy-wait loop until data values have arrived from all of its input sources; it then computes and writes results out to other processors. Parallel programming is simplified because much of the synchronization is accomplished implicitly by the underlying machine.

The data-flow execution mode does not eliminate synchronization as a problem of parallel computation, it only eliminates it as a problem for the programmer. The underlying hardware must still service the arrival

of data (asynchronously), determine when sufficient data has arrived to initiate processing, support queues for all of the input channels to hold the arriving data, and implement a "queue is full" signalling mechanism with the input data queues. These hardware facilities represent significant overhead and are incompatible with current efforts in the design of VLSI multiprocessors toward very simple processor structure.

In this paper, we consider the automatic conversion of data-flow programs into equivalent synchronous programs. Such conversions enable programmers to program as though the underlying machine executed in a data-flow mode, while allowing the hardware to execute synchronously. We begin with a model of parallel computation in which we can express both data-flow and synchronous computations. Within this model, we define a restricted class of programs and characterize the conditions under which a conversion from data-flow to synchronous execution is possible. Finally, we present two algorithms for performing the conversion: the first is more general but the second often produces better results. Although our algorithms apply only to a subclass of all parallel programs, it is sufficiently rich to encompass many of the recently developed parallel and systolic programs.

The Model of Parallel Programs

The formalism that we use to develop our algorithms and prove their correctness is quite spare. In order to connect it with conventional parallel computation settings, we give an informal description of the situation from which we have abstracted.

We postulate a parallel processor composed of m machines $M_1, M_2, \ldots, M_m$ which communicate with read and write operations. The

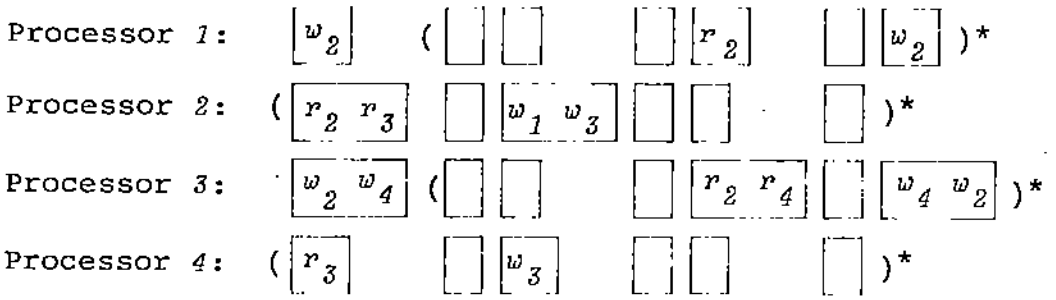
machines, referred to as processing elements or PEs, are all of the same type. In general, PEs will be sequential RAMs with small amounts of local memory (and no global memory) but it is sufficient to let them be devices capable of defining a regular set. This simplification is valid because we are concerned here only with a PE's interprocess input/output behavior and not its computational ability. We assume that the machines execute with a common time step; on each step a PE can attempt to perform a set of operations simultaneously. In synchronous mode, all operations will execute the first time that they are attempted. In data-flow mode, writes will execute as soon as they are attempted but, depending on the state, reads may block. A blocked operation is retried on the next execution step and a process does not proceed with a new set of operations until all of its current operations have completed.

We model such systems as *Interprocess Communication (IC) Systems*. An IC system is completely defined by a set of regular expressions, $V_1, V_2, \ldots, V_m$, each describing the interprocess input/output behavior of a single PE. The i-th regular expression describes the behavior of the i-th machine. The algorithms developed in this paper work for *loop programs* in which all regular expressions are of the form α^* where α is a sequence of symbols from the alphabet. We define ρ to be a function on expressions that removes the outermost Kleene star; $\rho(\alpha^*) = \alpha$. The symbols in our regular expressions denote sets of operations that are to be executed simultaneously. The alphabet is the power set of $\{r_i, w_i \mid i \in [m]\}$ [†] where r_j denotes a read from PE j, w_j denotes a write to PE j and $\{\}$ takes the place of any operation not involved in interprocess communication

---

[†] [m] denotes the set $\{1,2,3,\ldots,m\}$.

(including operations that transfer values to and from the external environment).† Figure 1(a) is an IC system representing the systolic



1(a) IC system representing systolic processor for band matrix - vector multiplication

1(b) Communication graph for the IC system of Figure 1(a)

Figure 1.

processor for band matrix-vector multiplication with a bandwidth of four [1]; only interprocess reads and writes appear in the model, all other operations are replaced by {}. Figure 1(b) shows the *communication graph* for this system; each vertex represents a PE and a directed edge from node i to node j represents a *communication link* over which the i-th PE writes to the j-th PE and the j-th PE reads from the i-th PE.

We define the execution of an IC system terms of two sequences, $C^1,C^2,C^3,\ldots$ and $T^0,T^1,T^2,\ldots$. Each element of the first sequence is an m-vector which gives the program counter values for all PEs (a program counter value is the index of a set of operations). Each element of the

† Note that we use standard set notation to represent both sets and the symbols of our alphabet; the distinction will be clear from the surrounding context. In our figures, we will use rectangular boxes to enclose sets rather than the usual brace notation.

second sequence is an $m \times m$ matrix of strings, giving the status of communications in terms of a generic message X. The status of communications on the link from PE i to PE j is given by $t_{i,j}$: $t_{i,j} = X^n$ means that there have been n unanswered writes; $t_{i,j} = (X^{-1})^n$ means that there have been n unanswered reads; and $t_{i,j} = \lambda$ means that there are no outstanding reads or writes (λ represents the null string). The sequences together describe the execution of a system; for all $k>0$, C^k describes the set of operations that will be attempted on the k-th execution step and T^k describes the status of communications if all of those operations complete.

To start the sequences, we define $c_i^1=1$ for all $i \in [m]$ and $t_{i,j}^0=\lambda$ for all $i,j \in [m]$; C^1 shows all PEs executing their first set of operations and T^0 shows that there are no outstanding reads or writes. The remainder of the sequence of Cs is defined to reflect the fact that a PE moves to a new set of operations only if all operations in its previous set have completed:

$$c_i^{k+1} = \begin{cases} c_i^k+1 & \text{if } \mathit{UNBLOCKED}(i, V_i(c_i^k), T^k) \\ c_i^k & \text{otherwise} \end{cases}$$

where the notation $V(j)$ denotes the j-th symbol in some word generated by the expression V [†] and $\mathit{UNBLOCKED}(i,S,T)$ is true if the i-th PE can execute all operations in set S when the status of communications is described by T. The exact form of *UNBLOCKED* depends on the mode of execution, synchronous or data-flow, and is discussed below. The remainder of the sequence of Ts is defined to reflect the execution of

---

[†] Note that for all loop programs, $V(j)$ is a unique symbol. This notation will also be used for processes that execute an initialization sequence before entering their loop. These PEs are represented by regular expressions of the form $\beta\alpha^*$ where α and β are sequences over the alphabet and, again, the j-th symbol is unique.

read and write operations:

$t_{i,j}^{k+1} = a \cdot t_{i,j}^{k} \cdot b$ where

$$a = \begin{cases} X & \text{if } w_j \in V_i(c_i^{k+1}) \wedge (k=0 \vee c_i^{k+1} \neq c_i^k) \\ \lambda & \text{otherwise} \end{cases}$$

and

$$b = \begin{cases} X^{-1} & \text{if } r_i \in V_j(c_j^{k+1}) \wedge (k=0 \vee c_j^{k+1} \neq c_j^k) \\ \lambda & \text{otherwise} \end{cases}$$

and $X \cdot X^{-1} = \lambda$. We observe that our execution rules are more general and more realistic than those used in many models because we do not insist that all of the operations in a set execute simultaneously. Depending on the definition of *UNBLOCKED*, it is possible, for example, to allow independent reading and writing on different ports.

The execution of an IC system is parameterized by the predicate *UNBLOCKED*. When the predicate is *TRUE*, the IC system is *synchronous*, that is, all operations execute on every time step. A correct, synchronous system should have the property that corresponding reads and writes are simultaneous.† More precisely, if during synchronous execution, $t_{i,j}^k = \lambda$ for all i, j and k, we say that the system is *strongly coordinated*. When the predicate *UNBLOCKED*(i,S,T) is

$$\forall j \in [m]\ (r_j \in S \Rightarrow t_{i,j} \in X^*)$$

the IC system is *data flow*, that is, read operations execute only when values are present. A correct, data-flow program should have the property that none of the individual PEs deadlock. We say that a system is *valid* if

† It is more common to assume that a read executes immediately after its corresponding write. We have chosen simultaneous reads and writes to be consistent with VLSI technology and to simplify our discussion. All of our algorithms can be easily modified to incorporate any fixed delay for message transmission.

$$\forall i \in [m] \forall k \geq 0 \; \exists j > k \; (c_i^k \neq c_i^j)$$

when the system is executed in data-flow mode.

We remark that the model developed here differs from the well-known vector addition system model [2] and the Petri Net model [3]. In the VAS model, there is a specific execution mode: transition vectors are applied only if all relevant coordinates are positive and when a transition vector is applied, all coordinates are updated simultaneously. There is also a specific execution mode for Petri Nets: transitions fire only if all incident places contain a token and all token values are updated simultaneously. In contrast, IC systems may execute in either synchronous or data-flow mode. In synchronous mode, operations execute as soon as they are attempted. In data-flow mode, execution is conditional on the appropriate values being available as in the VAS and Petri Net models. However, even in data-flow mode, our model differs from the other two since operations execute whenever they are enabled and the input and output of an instruction are not necessarily simultaneous.

Variants

We would like to convert data-flow programs into strongly coordinated, synchronous programs. For such algorithms to be useful, the resulting program must perform the same computation as the original program. To make this more precise, we define the notion of the set of reads preceding a specific write. Writes, in data-flow mode, execute on the first step in which they are attempted; the set of writes executed by PE i in execution step k, $WRITES(i,k)$, is

$$\{ w_j \mid w_j \in V_i(c_i^k) \wedge (k=1 \vee c_i^k \neq c_i^{k-1}) \}.$$

Reads, in data-flow mode, may block temporarily and so a read executes in the first step in which it was attempted and the corresponding data was available; the set of reads that PE i executes in step k, $READS(i,k)$, is

$$\{\, \tau_j \mid \tau_j \in V_i(c_i^k) \wedge t_{j,i}^k \neq X^{-1} \wedge ((t_{j,i}^{k-1} = X^{-1}) \vee ((k=1 \vee c_i^k \neq c_i^{k-1}) \wedge (t_{j,i}^k = t_{j,i}^{k-1} \cdot X^{-1} \vee w_i \in WRITES(j,k)))) \,\} .$$

This means that a read in the current operation set executes on step k if it is no longer pending after k ($t_{j,i}^k \neq X^{-1}$) and one of three conditions is met: it had been pending in the previous step ($t_{j,i}^{k-1} = X^{-1}$); or it was first attempted in step k ($k=1 \vee c_i^k \neq c_i^{k-1}$) and there were unanswered writes available ($t_{j,i}^k = t_{j,i}^{k-1} \cdot X^{-1}$); or it was first attempted in step k and a corresponding write also occured in step k ($w_i \in WRITES(j,k)$).

The l-th write from PE i to PE j occurs on execution step k such that

$$l = \sum_{p=1}^{k} x_p \quad \text{where} \quad x_p = \begin{cases} 1 & \text{if } w_j \subset WRITES(i,p) \\ 0 & \text{otherwise} \end{cases}$$

and the set of reads that precede that l-th write, $PREADS(l,i,j)$, is the multi-set $\bigcup_{p=1}^{k-1} READS(i,p)$. From this, we can define the relationship that we wish to hold between the original data-flow system and our constructed, synchronous system.

In terms of our abstraction, we will say the constructed system P' performs the same computation as the original system P if three requirements are met. The first requirement is that a PE communicates with the same set of PEs in both systems. Our second requirement is that there is at least as much data available to a PE at the time of any write in P' as there was available in P. This second requirement will be true if the set of reads that precede any write in P is a subset of the set of reads that precede that same write in P'. Thus, we allow reads to occur "ear-

lier" and writes "later" in the constructed system than they did in the original system; we assume that resulting, additional data is buffered within the PE. To insure that the PEs remain finite, our third requirement is that the amount of this additional buffering is bounded. Putting this together, we say the new system P' is a *variant* of the original system P if

(i) they have the same communication graphs

(ii) for each pair of PEs i and j and for all $l \geq 0$

$$PREADS(l,i,j) \subseteq PREADS'(l,i,j)$$

and

(iii) there is some b such that for each pair of PEs i and j and for all $l \geq 0$

$$|PREADS'(l,i,j) - PREADS(l,i,j)| \leq b \quad .$$

We present the following propositions without proof

Proposition 1: The relation "variant of" is transitive.

Proposition 2: If $P = V_1, V_2, \ldots, V_m$ is a valid, loop program and $n_1, n_2, \ldots, n_m$ are integers greater than 0, then

$$V_1^{n_1}, V_2^{n_2}, \ldots, V_m^{n_m}$$

is a variant of P.

The problem that we consider in the remainder of this paper can now be formally stated:

Given a valid, data-flow loop program, construct a strongly coordinated variant.

The Coordination Problem

The coordination problem cannot be solved for all data-flow, loop programs. Consider, for example, the system in Figure 2. We define the

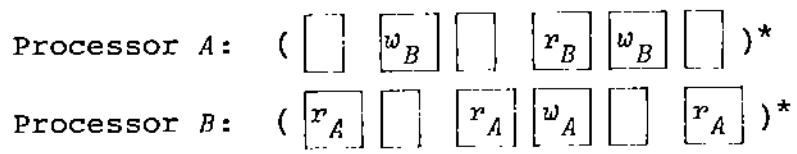
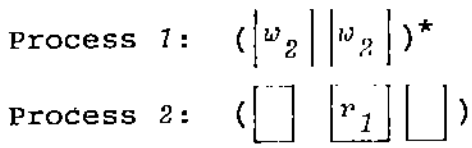


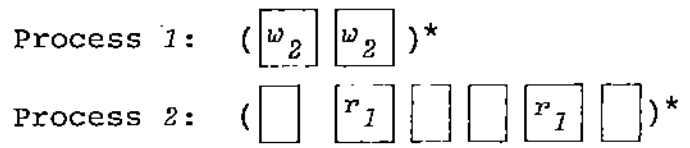
Figure 2. An IC system that has no balanced variant

rate at which a PE uses a communication link to be the number of reads or writes by that PE to the link in one cycle of its execution. The PEs in the example communicate across the link from B to A at the same rate but they communicate across the link from A to B at different rates. Intuitively, to strongly coordinate this system, the cycles of A must "speed up" relative to the cycles of B. Any speed up of A, however, causes the communication rates across the link from A to B to differ. This new mismatch can only be corrected by speeding up the cycles of B relative to the cycles of A, returning us to the original problem. There is no strongly coordinated variant of the system in Figure 2. The problem with the system is not simply a matter of unmatched data rates: the data rates across the link of the system in Figure 3(a) are also unmatched but the system has a strongly coordinated variant shown in Figure 3(b). The distinction between systems that can be coordinated and systems that cannot be coordinated is more subtle.

Defining $ON(i,j)$ to be the number of writes by PE i to PE j in V_i and $OFF(i,j)$ to be the number of reads by PE j from PE i in V_j, we say that a



3(a) An unbalanced system



3(b) A balanced variant of the system in Figure 3(a)

Figure 3.

system is *balanced* if the following set of *balancing equations* has a solution in which all $x_i=1$

$$\{ON(i,j)\cdot x_i = OFF(i,j)\cdot x_j | i,j \in [m]\} .$$

Neither the system in Figure 2 nor the system in Figure 3(a) are balanced; the system in Figure 3(b) is balanced.

Lemma *1:* There is at most one independent, non-trivial solution to a set of balancing equations.

Proof: Form a spanning tree T of the undirected graph underlying the communication graph for the given system. Each vertex of T corresponds to a variable in the balancing equations and each edge of T corresponds to one of the equations. It is sufficient to show that each of the $(m-1)$ corresponding equations are independent. Consider a variable x_i corresponding to a leaf node of T. There is exactly one edge e to the node corresponding to x_i and so there is one equation represented by T that uses x_i. That equation must be independent of the other $(m-2)$ equations represented by T-$\{e\}$ and by induction those equations must be independent of each other. //

A loop program is said to be *balancable* if its balancing equations have a solution in which all of the x_i are integers greater than 0. The system in Figure 2 is not balancable because there is no nontrivial solution to the set of equations

$$2 \cdot x_A = 3 \cdot x_B$$
$$x_B = x_A \ .$$

The system in Figure 3(a) is balancable because $x_1 = 1$ and $x_2 = 2$ is a solution to the equations. If a loop program $V_1, V_2, \ldots, V_m$ is balancable then a solution to the balancing equations $x_1, x_2, \ldots, x_m$ can be found in $O(n^3)$ time and by Proposition 2, we can construct a balanced variant, $V_1', V_2', \ldots, V_m'$ by setting each $V_i' = (\rho(V_i)^{x_i})^*$.

We can now state the relationship between loop programs which can be strongly coordinated and balancable programs.

Theorem 1: A valid, loop data-flow program can be strongly coordinated if and only if it is balancable.

Proof:

(<=) This proof is given later as the proof of our Wave Algorithm.

(=>) Let P be a valid data-flow program and let P' be a strongly coordinated variant of P. Because P' contains only loop programs, it is possible to consider the c values for any PE i as integers modulo the length of V_i. With this change in program counter values, P' is finite state since there is no buffering of transmitted values. Therefore there is some state, q, which appears infinitely often in the execution of P' and the execution sequences appearing between any two consecutive occurences of q must be the same. Consider an arbitrary PE i, and let O be the multi-set of operations it executes in a single cycle and let E be the multi-set of operations it executes as the system moves from one occurence of q to

the next. Let O^n be n-fold union of O.

Claim: $E = O^n$ for some $n \geq 1$.

Proof: Suppose not. Then let $Y = E - O^r$ where r is the greatest integer such that $O^r \subseteq E$. Y is the set of "extra" operations that do not form a complete cycle. Suppose there is some write operation, w_j, in Y. Then Y must contain all of the read operations in O as well, since otherwise the writes to PE j would "move up" relative to the reads by PE i and eventually, for some k, we would have

$$PREADS'(k,i,j) \subset PREADS(k,i,j) \ .$$

Suppose there is a read, r_j, in Y. Then Y must also contain all of the write operations in O as well, since otherwise the reads would "move up" relative to the writes by PE i and, for any bound b there would be some k for which

$$|PREADS'(k,j,i) - PREADS(k,j,i)| > b.$$

Unless Y is empty, we have a contradiction. This completes the proof of the claim.

So for each PE, $E = O^n$ for some integer n. Choosing x_i to be the appropriate value of n for PE i, the x_i's form the desired solution to the balancing equations. //

The class of programs that can be strongly coordinated is quite large and it includes for example most of the systolic and pipelined algorithms. As another characterization, an IC system has the *finite buffer property* if, when executed in data-flow mode, there is some integer b such that for all i, $j \in [m]$ and $k \geq 0$, $t^k_{i,j} \leq b$. This is obviously a desirable characteristic for any data-flow program and we show

Theorem 2: Any valid, loop program with the finite buffer property can be strongly coordinated.

Proof: If D is a valid, loop program with the finite buffer property, then, as above, there must be some state which repeats infinitely often in the execution sequences for D. Between every two consecutive occurences of this state in a sequence, a PE must execute and integral number (greater than 0 because the system is valid) of its cycles and data rates onto and off of each communication link must be equal. As a result, if we set x_i to the number of cycles PE i executes during this sequence, then the x_i's form a solution to the balancing equations in which all $x_i > 0$. //

From this theorem and the example in Figure 3(a), we can conclude

Corollary: The set of valid, loop programs with the finite buffer property is properly contained in the set of valid, loop programs that can be strongly coordinated.

In the next section of this paper, we present our algorithms for converting data-flow programs into a strongly coordinated programs. The algorithms work only for balancable, valid loop programs. We have shown how to determine whether or not a program is balancable, now we show how to determine whether or not it is valid.

Theorem 3: If a loop program is balancable, then there is an efficient method for testing its validity.

Proof: Let S be a balancable loop program and let B be its balanced variant constructed as above. The words generated by each PE have not been changed in B, so S is valid if and only if B is valid. If $B = V_1, V_2, \ldots, V_m$, construct the system $D = \rho(V_1)\{\}^*, \rho(V_2)\{\}^*, \ldots, \rho(V_m)\{\}^*$.

Claim: B is valid if and only if D is valid.

Proof:

(=>) Immediate since B is balanced.

(<=) Suppose that D is valid and B is not valid. Then there must be some subset of the PEs, $p_1, p_2, \ldots, p_n$, which become circularly blocked; that is, for all $i \in [n]$, PE i blocks on a read from its successor, that is, PE $(i \bmod n)+1$. Consider the first point in the execution sequence at which this circular blocking occurs. Since B is balanced, the PEs must all be on the same iteration of their cycles at this point and within this iteration, the number of writes to PE i by its successor must be one less than the number of reads by PE i from its successor. For the circular blocking to arise in B, it must be that for all $i \in [n]$, the read which blocks in PE i must come before the write that releases its predecessor. But as a single PE executes in B or D, its reads retain the same position relative to its writes so this must be true in D as well. If the blocked reads all precede the releasing writes in D, however, then D would block on the same operations. This is a contradiction since D is valid, completing the proof of the claim.

D can be tested for validity by executing it until it reaches a step k for which for all i, $c_i^k > |\rho(V_i)| \vee \forall l > k\,(c_i^k = c_i^l)$. Once such a stable state has been reached, the validity of D can be tested by determining whether or not all read and write operations have completed. If s is the number of operations executed by PEs in a single cycle of S, D will execute for at most s steps and so the test requires $O(s)$ time. //

The Conversion Algorithms

In this section, we provide algorithms for automatically converting a data-flow loop program into a strongly coordinated variant when possible.

For an arbitrary program P, we start by constructing a balanced variant and testing it for validity. If P is balancable and valid, then its balanced variant is coordinated with one of the two algorithms presented in this section. Proposition 1 insures that the resulting, strongly coordinated system is a variant of P.

Starting with a balanced, valid variant, we construct a strongly coordinated variant with the following algorithm.

Algorithm 1: Wave algorithm to coordinate loop data-flow programs

Input: A valid, balanced, loop program, $V_1, V_2, \ldots, V_m$

Output: A strongly coordinated variant of the given program, $V_1', V_2', \ldots, V_m'$

Method:

1. Form expressions $R_1, R_2, \ldots, R_m$ from the given expressions where $R_i = \rho(V_i)(\|)^*$.
2. Compute the data-flow execution sequences $C^1, C^2, \ldots, C^k$ and $T^0, T^1, \ldots, T^k$ where k is the least integer for which $c_i^k > |\rho(V_i)|$ for all i.
3. For each i and for $l = 1,2,\ldots,k$, set $V_i'(l)$ to $READS(i,l) \cup \{w_j \mid r_i \in READS(j,l)\}$.

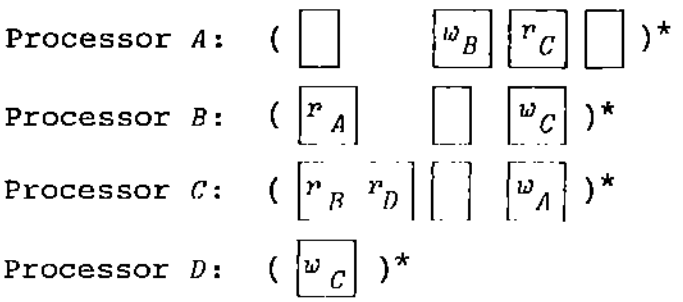
Theorem 4: The Wave Algorithm constructs a strongly coordinated variant of any valid, balanced, loop program.

Proof: Since the original system is valid, we are assured of finding a value for k in step 2. By the construction in step 3, writes can only occur in the same step as their corresponding reads so the system is strongly coordinated (the complete justification of this appears in a paper on testing

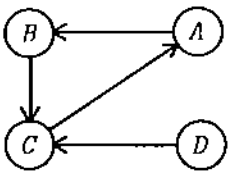
coordination properties [4]). It remains to show that the constructed system is a variant of the original system. Since the set of reads executed in any cycle of a PE is the same in both the given and constructed systems, requirements (*i*) and (*iii*) for a variant are trivially satisfied and, for requirement (*ii*), it is sufficient to consider just the first k execution steps of the system. For $l=1,2,\ldots,k$, it is obvious that $READS(i,l) = READS'(i,l)$ for all i where READS is defined for the execution sequences of the given system and READS' is defined for the execution sequences of the constructed system. Suppose the second requirement is violated by the l-th write from PE i to PE j which occurs on the r-th step of the execution sequence for the constructed system and the s-th step of the execution sequence for the given system where $r<s$. The write in the constructed system occurs in the same step as its corresponding read in both systems. Therefore in the original system the read that corresponds to the write in step s must occur in step r (before s), which is not possible by the definition of data-flow execution. //

If s is the total number of operations executed by PEs in a single cycle of $V_1', V_2', \ldots, V_n'$, then $k \leq s$ and for all i, $|\rho(V_i')| \leq s$. The algorithm builds each symbol of each V_i' and so it requires $O(ms)$ time.

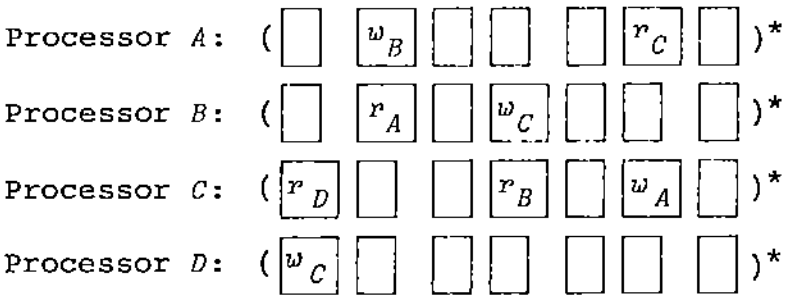
Figure 4 is an example of a valid, data-flow system and its strongly coordinated variant constructed by this algorithm. The name of the algorithm comes from the fact that a single cycle's data passes through the entire system before any PE starts its next cycle. For this example, the result is nearly optimal because the data dependencies of the program do not allow any of the PEs to get more than a few operations ahead of the remaining PEs. However, if the original system is changed even slightly, as in Figure 5, the result is unsatisfactory. In this case, a better solution



Data flow program



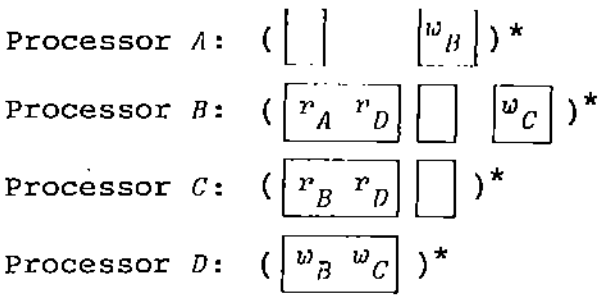
Communication graph



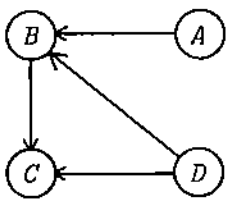
Strongly coordinated variant constructed by the Wave Algorithm

Figure 4.

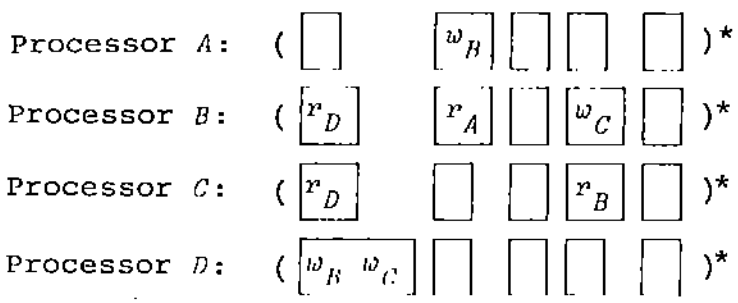
is to allow Processors A and D to start a full cycle ahead of the others. After they have completed their first cycles, Processor B can begin executing its first cycle, while Processors A and D continued with their second cycles. By the third cycle of A and D, all processors are executing on every step. This more efficient solution, pictured in Figure 6, maintains the original three step cycle for the processors. The writes have been moved "forward" so that, for example, the write which occurs at the beginning of the second cycle for Processor A is delayed from its first cycle. The w_B in the third cycle of Processor D is delayed from its second cycle and the w_C in the third cycle of Processor D is delayed from its first cycle. The solution was constructed by the following coordination algorithm for systems with acyclic communication graphs.



Data flow program

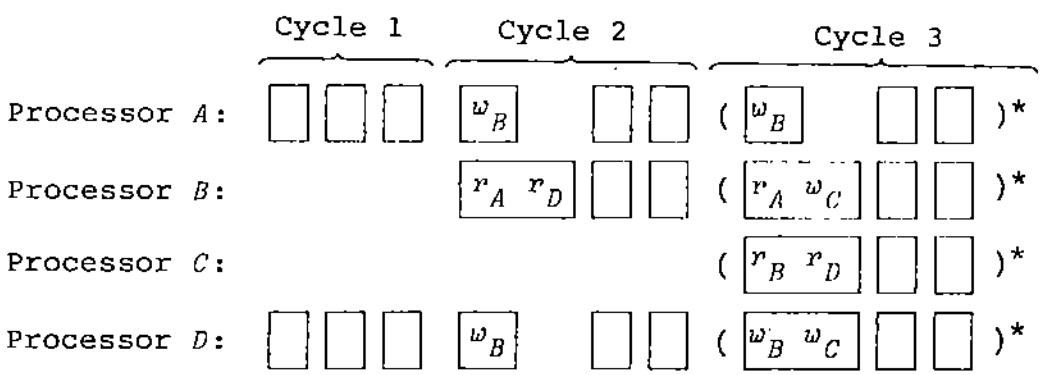


Communication graph



Strongly coordinated variant from Wave Algorithm

Figure 5.



Strongly coordinated variant of IC system from Figure 5.

Figure 6.

Algorithm 2: Buffered Write Algorithm

Input: A valid, balanced, loop program, $V_1, V_2, \ldots, V_m$ with an acyclic communication graph.

Output: A strongly coordinated variant of the given system, $V_1', V_2', \ldots, V_m'$.

Method:

1. Label the nodes of the communication graph with the length of the longest path from a source node (a node with no predecessors) to the node. Let *LMAX* be the depth of the graph.
2. Let n be the maximum length of any $\rho(V_i)$ and form the expressions $R_1, R_2, \ldots, R_m$ where for all i

$$R_i = \{\}^{nl} \; (\; E \cdot (\{\})^{n - |\rho(V_i)|} \;)^{LMAX - l}$$

   where l is the label of the node for PEi and the expression E is $\rho(V_i)$ with all of the write operations removed (if writes are the only operations on some step, replace them with $\{\}$).
3. For each $i \in [m]$, for each $k = 1$ to $|\rho(R_i)|$, set $V_i'(k)$ to

$$R_i(k) \cup \{ \; w_j \; | \; r_i \in R_j(k) \}$$

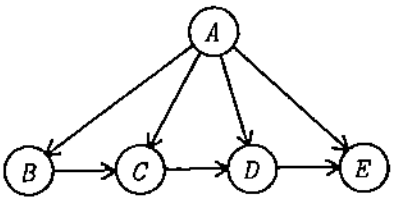
Theorem 5: The Buffered Write Algorithm constructs a strongly coordinated variant for any valid, balanced, loop program that has an acyclic communication graph.

Proof: The fact that the resulting program is strongly coordinated follows the construction in step 3 as above, so it remains to show that it is a variant of the given system. Conditions (*i*) and (*iii*) for variance are obviously met, leaving condition (*ii*). The execution sequence for the output system can be divided into periods equal in length to the cycle size. Consider a single PE which both reads and writes (if a PE does not both read and write, it trivially satisfies the second condition). After the initial

periods in which the PE just idles, it executes all of the reads from one cycle during each period. Therefore, it is sufficient to show that if the reads from a given iteration of the PE occur during period k, then the writes for that iteration occur no sooner than period $k+1$. Because the communication graph is acyclic, this is easily done by induction on the periods noting that (1) the writes executed during any period are a subset of the writes for a cycle and (2) the first read occurs at least one period before the first write. //

The R_i will have a common length n equal to *LMAX* times l. In order to set the value of each symbol in each one of the V_is, all of the symbols in the corresponding position of the R_is must be examined and so the algorithm runs in $O(m^2n)$ time.

This algorithm works for all acyclic loop programs but it does not always produce a good solution. Consider the system in Figure 7. The Buffered Write construction creates a long initialization sequence (the maximum length is the maximum cycle size times the number of PEs) which means that many of the PEs idle for long times and that the length of the PE code increases. The extra idling is probably not significant since we can assume in most cases that the number of PEs will be much smaller than the number of iterations required. The longer code, however, is a more serious problem since PEs will normally have a very limited amount of memory. For this example, a better solution is the coordinated program in Figure 8 in which each of the writes has simply been moved "forward" two steps. Because the movement was within one cycle, the PEs do not have to stagger their starts. The Buffered Write Algorithm can be modified to produce this code by "preprocessing" the communication graph to eliminate links for which all writes appear before their



Process A: ($w_B\ w_C\ w_D\ w_E$ |)*

Process B: (w_C | r_A)*

Process C: (w_D | $r_A\ r_B$)*

Process D: (w_E | $r_A\ r_C$)*

Process E: (| $r_A\ r_D$)*

Strongly coordinated variant constructed by
Buffered Write Algorithm

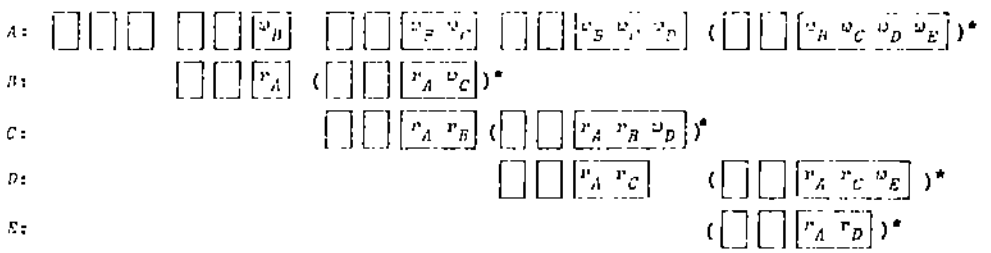


Figure 7.

corresponding reads.

As a final comment, notice that the compute operations have been completely ignored in our analysis. To be realistic, we would have to argue that our notion of variant preserves the computations of the system. In fact, our definition does not preserve the computations of the system since it does not preserve the order of compute steps or the

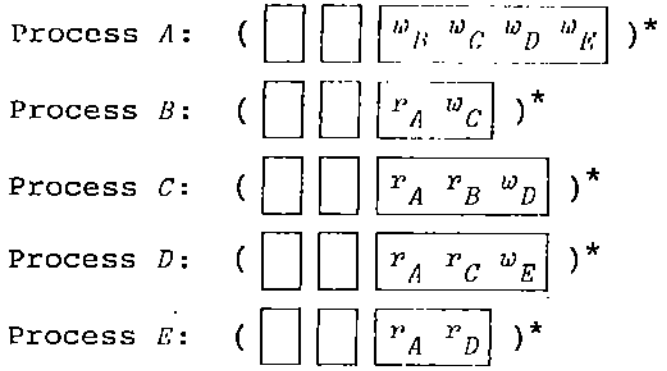
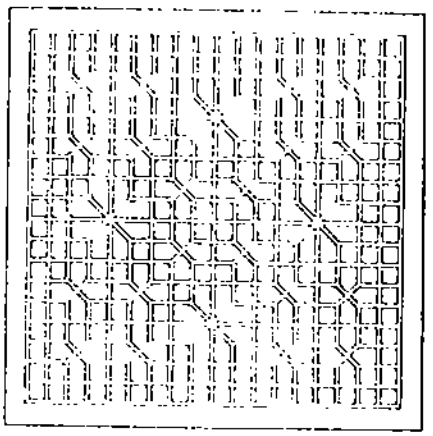


Figure 8. Strongly coordinated variant of the system in Figure 7.

information available to a PE on a compute step. The definition of variant would have to be strengthened. It should be noted, however, that our algorithms could be easily adapted to this stronger definition since they retain the position of all compute steps.

Conclusions

We have presented a simple model of parallel computation in which both data-flow and synchronous execution modes can be harmoniously expressed. Given certain programs defined using the data-flow execution mode, we have shown how to synthesize programs that are computationally equivalent when executed in the synchronous mode. For the class of programs under consideration, we characterized those for which this synthesis is possible using the concept of "balancable". Potentially, our algorithms can be used to shift the burden of specifying detailed timing behaviors from the programmer to a compiler.



Octagon Lattice, Mount Omei,
Szechwan, (1800-1900 A.D.)

Acknowledgments

We owe a debt of gratitude to Dennis Gannon for useful discussions concerning coordination and for the proof of Lemma 1.

References

[1] H.T. Kung and C.E. Leiserson, Systolic arrays (for VLSI), Technical Report CS-79-103, Carnegie Mellon University (1979).

[2] Richard Karp and R.E. Miller, Properties of a model for parallel computations: determinacy, termination, queuing, SIAM J. Appl. Math. 14:6, pp.1390-1411 (1966).

[3] James L. Peterson, Petri Nets, Comp. Surveys 9:3, pp.223-252 (1977).

[4] Janice E. Cuny and Lawrence Snyder
Testing the coordination problem
Technical Report CSD-TR-391, Purdue University, 1982 (in preparation).